Appears in the Proceedings of the 2nd Conference on File and Storage Technologies (FAST'03). pp. 29–42 (31 Mar – 2 Apr 2003, San Francisco, CA). Published by USENIX, Berkeley, CA

Plutus: Scalable secure file sharing on untrusted storage

Mahesh Kallahalla* Erik Riedel† Ram Swaminathan* Qian Wang‡ Kevin Fu§

Hewlett–Packard Labs
Palo Alto, CA 94304

Abstract

Plutus is a cryptographic storage system that enables secure file sharing without placing much trust on the file servers. In particular, it makes novel use of cryptographic primitives to protect and share files. Plutus features highly scalable key management while allowing individual users to retain direct control over who gets access to their files. We explain the mechanisms in Plutus to reduce the number of cryptographic keys exchanged between users by using filegroups, distinguish file read and write access, handle user revocation efficiently, and allow an untrusted server to authorize file writes. We have built a prototype of Plutus on OpenAFS. Measurements of this prototype show that Plutus achieves strong security with overhead comparable to systems that encrypt all network traffic.

1 Introduction

As storage systems and individual storage devices themselves become networked, they must defend both against the usual attacks on messages traversing an untrusted, potentially public, network as well as attacks on the stored data itself. This is a challenge because the primary purpose of networked storage is to enable easy sharing of data, which is often at odds with data security.

To protect stored data, it is not sufficient to use traditional network security techniques that are used for securing messages between pairs of users or between clients and servers. Thinking of a stored data item as simply a message with a very long network latency is a misleading analogy. Since the same piece of data could be read by multiple users, when one user places data into a shared storage system, the eventual recipient of this "message" (stored data item) is often not known in advance. In addition, because multiple users could update the same piece of data, a third user may from time-to-time update "the message" before it reaches its eventual recipient. Stored data must be protected over longer periods of time than typical message round-trip times.

Most existing secure storage solutions (either encrypt-on-wire or encrypt-on-disk [40]) require the creators of data to trust the storage server to control all users' access to this data as well as return the data intact. Most of these storage systems cater to single users, and very few allow secure sharing of data any better than by sharing a password.

This paper introduces a new secure file system, *Plutus*, which strives to provide strong security even with an untrusted server. The main feature of Plutus is that all data is stored encrypted and all key distribution is handled in a decentralized manner. All cryptographic and key management operations are performed by the clients, and the server incurs very little cryptographic overhead. In this paper we concentrate on the mechanisms that Plutus uses to provide basic filesystem security features — (1) to detect and prevent unauthorized data modifications, (2) to differentiate between read and write access to files, and (3) to change users' access privileges.

Plutus is an encrypt-on-disk system where all the key management and distribution is handled by the client. The advantage of doing this over existing encrypt-on-wire systems is that we can (1) protect against data leakage attacks on the physical device, such as by an untrusted administrator, a stolen laptop, or a compromised server; (2) allow users to set arbitrary policies for key distribution (and therefore file sharing); and (3) enable better server scalability because most of the computationally intensive cryptographic operations are performed at end systems, rather than in centralized servers.

By using client-based key distribution, Plutus can allow user-customizable security policies and authentication mechanisms. Relative to encrypt-on-wire systems, clients individually incur a higher overhead by taking on the key distribution, but the aggregate work within the system remains the same. Our previous analysis with fi-

* {maheshk, swaram}@hpl.hp.com.

† Work done while at HP Labs. Current address: Seagate Technology, Pittsburgh, PA 15222; erik.riedel@seagate.com.

‡ Work done while at HP Labs. Current address: Mechanical Engineering Department, Pennsylvania State University, University Park, PA 16802; quw6@psu.edu.

§ Work done while at HP Labs. Current address: Laboratory for Computer Science, MIT, Cambridge MA 02139; fubob@mit.edu.

legroups [40], which aggregates keys for multiple files, shows that the number of keys that any individual needs can be kept manageable.

Instead of encrypting and decrypting files each time they are exchanged over the network, Plutus pre-computes the encryption only when data is updated; this is a more scalable solution as the encryption and decryption cost is distributed among separate users and never involves the server.

We have built a prototype of Plutus in OpenAFS [37]. This enhancement to OpenAFS retains its original access semantics, while eliminating the need for clients to trust servers. Measurements on this prototype show that strong security is achievable with clients paying cryptographic cost comparable to that of encrypt-on-wire systems, and servers not paying any noticeable cryptographic overhead. Since the cryptographic overhead is shifted completely to the clients, the server throughput is close to that of native OpenAFS. Note that these modifications have no impact on the way end applications access files; they change only the way users set sharing permissions on files.

The rest of the paper is organized as follows. Section 2 describes our threat model and assumptions. Section 3 presents the mechanisms and design of Plutus. Section 4 describes a number of subtle attacks that remain possible and outlines potential solutions, and Section 5 describes protocols for creating, reading and writing files, and revoking users. Section 6 describes the implementation and usage of Plutus, and Section 7 evaluates the prototype. We discuss related work in Section 8 and conclude in Section 9.

2 Threat model

This section discusses the assumptions and threat model of Plutus. This paper will use the terminology introduced previously [40] with *owners* (create data), *readers* (read data), *writers* (write and possibly read data), and *servers* (store data).

2.1 Untrusted servers and availability

In Plutus, we trust servers to store data properly, but not to keep data confidential. While a server in Plutus may attempt to change, misrepresent, or destroy data, clients will detect the malicious behavior.

Cryptography alone, however, cannot prevent destruction of data by a malicious server. Replication on multiple servers can ensure preservation of data even when many of the servers are malicious. Systems such as BFS [7], Farsite [1], OceanStore [25], PASIS [17], PAST [12], and S4 [47] address techniques for secure availability through replication. Though, in this paper, we restrict our focus to securing data on a single untrusted file server, the ideas could be generalized for a set of replicated file servers.

2.2 Trusted client machine

Users must trust their local machine. This is, however, difficult to guarantee: providing for a secure program execution environment in an untrusted computing platform is an open problem. Some previous work aims to securely monitor loaded applications [48] or provide partitioned virtual machines to isolate vulnerabilities [10, 48, 50].

2.3 Lazy revocation

Plutus allows owners of files to revoke other people's rights to access those files. Following a revocation, we assume that it is acceptable for the revoked reader to read unmodified or cached files. A revoked reader, however, must not be able to read updated files, nor may a revoked writer be able to modify the files. Settling for lazy revocation trades re-encryption cost for a degree of security. We elaborate on lazy revocation in Section 3.4.

2.4 Key distribution

We assume that users authenticate each other to obtain relevant keys to read and write data on the disk via a secure channel – we do not introduce new authentication mechanisms in this paper. Furthermore, all these exchanges are carried out on-demand; if users want to read/write a file, they contact the file owner (or possibly other readers/writers) to obtain the relevant key. Keys are never broadcast to all users.

2.5 Traffic analysis and rollback

We do not address the issue of traffic analysis in this paper; that is, we do not make any explicit attempt to obfuscate users' access patterns. However, Plutus does support options to encrypt filenames, and encrypts all I/O requests to the server. Recently SUNDR [32] introduced the notion of a rollback attack, wherein an untrusted server tricks a user into accepting version-wise inconsistent or stale data relative to other users. We defer the discussion of rollback protection to a future work [15].

3 Design

In an encrypted file system, we need techniques to (1) differentiate between readers and writers; (2) prevent destruction of data by malicious writers; (3) prevent known plaintext attacks with different keys for different files; (4) revoke readers and writers; and (5) minimize the number of keys exchanged between users. The following core

mechanisms together achieve these functions: filegroups, lockboxes, keys, read-write differentiation, lazy revocation, key rotation, and server-verified writes.

3.1 Filegroups and lockboxes

Plutus groups files (not users) into *filegroups* so that keys can be shared among files in a filegroup without compromising security. Filegroups serve as a file aggregation mechanism to prevent the number of cryptographic keys a user manages from growing proportional to the number of files[1]. Aggregating keys into filegroups has the obvious advantage that it reduces the number of keys that users need to manage, distribute, and receive. This is important if users have to perform all key management and distribution themselves. Key aggregation is also necessary to support semi-online users: as in today's systems, Plutus assumes that users are frequently online, but not always. This means that we need an easy mechanism to let an owner share a group of related files, so that the other user may be able to access the related files even when the owner is not online. Additionally, as described in Section 3.2, we associate a RSA key pair with each filegroup. If files were not aggregated and each file had its own key pair, from the measurements in Section 7, each create operation would incur a 2.5 seconds latency to generate the RSA key pair – in comparison, it takes 2.9 seconds to encrypt/decrypt a 20M file with 3DES.

With filegroups, all files with identical sharing attributes are grouped in the same filegroup and are protected with the same key. This exploits the fact that even though a user typically owns and accesses many files, the number of equivalence classes of files with different sharing attributes is small; this enables multiple files to share the same set of keys.

Using filegroups dramatically reduces the number of keys that a user needs to keep track of and the number of keys users must obtain from other users. In the context of the sharing semantics of current UNIX file systems, if two files are owned by the same owner, the same group, and have the same permission bits, then they are authorized for access by the same set of users. All such files could logically be placed in the same filegroup, and encrypted with the same key.

In general there is no relation between the directory hierarchy and the files in a filegroup, though it may be sometimes convenient to define filegroups based on the set of files in one directory (which is, for instance, how AFS defines access rights). Specifically, two encrypted files from two different directories may belong to the same filegroup. Thus, filegroups can be viewed as an invisible overlay on the directory structure.

[1] A previous study [40] mistakenly attributes the filegroup concept to Cepheus [13] instead of itself.

Filegroups uniquely identify all keys that a user needs to perform an operation on a file. This filegroup information can be located together with the rest of the meta-data about the file, for instance, in the UNIX FFS inode (replacing the group and mode bits), or by adding an entry in the disk vnode in AFS [43].

On the downside, using the same key to encrypt multiple files has the disadvantage that the same key encrypts more data, potentially increasing the vulnerability to known plaintext and known ciphertext attacks. However, this is not an issue if these keys are actually the *file-lockbox* keys, and the real file encryption keys are different for different files. The lockbox can then securely hold the different keys; Section 3.3 explains further.

Filegroups also complicate the process of revoking users' access to files because now there are multiple files that the revoked user could have access to. It is tempting to simplify revocation of users by having one key per file. Though this scheme is seemingly more secure (losing a key compromises one file only), managing these keys is a challenge. At best they can be organized into some sort of hierarchy such that the users have to keep fewer keys securely, but this clearly resembles filegroups. Plutus' solution for this problem is discussed in more detail in Section 3.4.

3.2 Keys

Figure 1 illustrates the different objects in Plutus, and how different keys operate on them. Here we describe the structures; later sections discuss these design decisions in more detail. Every file in Plutus is divided into several blocks, and each block is encrypted with a unique symmetric key (such as a DES key), called a *file-block key*. The lockbox, based on ideas in Cepheus [13], holds the file-block keys for all the blocks of the file and is read and written by *file-lockbox* keys. File-lockbox keys are symmetric keys and are given to readers and writers alike. Alternatively, Plutus could use a single file-block key for all blocks of a file and include an initialization vector. File-lockbox keys are the same for all the files in a filegroup. In order to ensure the integrity of the contents of the files, a cryptographic hash of the file contents is signed and verified by a public-private key pair, which we call *file-verify keys* and *file-sign keys*. The file-sign keys are the same for all the files in a filegroup. As an optimization, a Merkle hash tree [34] is used to consolidate all the hashes, with only the root being signed.

Unlike files, which are encrypted at the block level, entries of directories are encrypted individually. This allows the server to perform space management without involving the clients, such as allocating inodes and performing a fsck after a crash. Also, this allows users to browse directories and then request the corresponding keys from

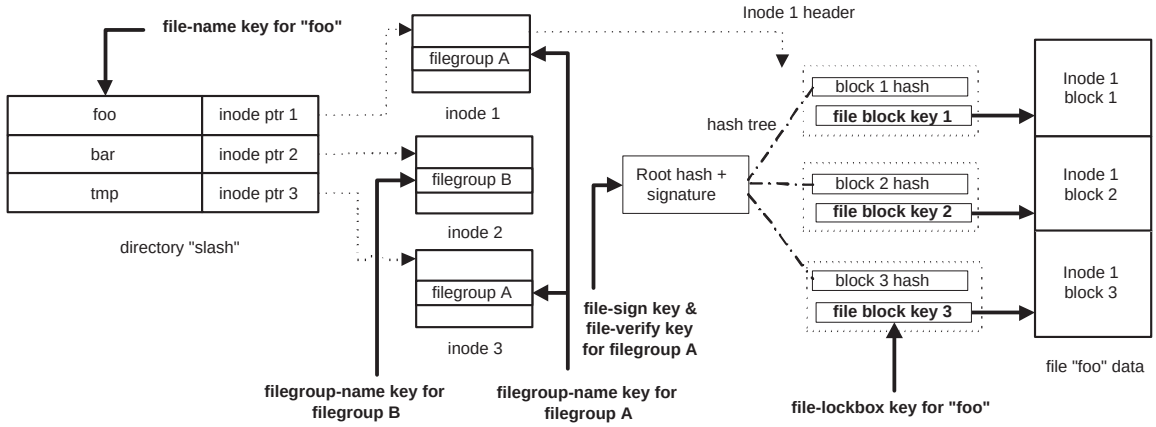

Figure 1: Keys in Plutus. The keys are all highlighted in bold and are linked to the objects that they operate on using bold lines. Dashed lines indicate object pointers. *File-name keys* can encrypt the names of files in directories. An inode contains the names of the filegroup that the file belongs to, and the *filegroup-name key* can encrypt filegroup names. The header contains the Merkle hash tree. The leaves of the hash tree are lockboxes containing the *file-block keys*, which are encrypted with the *file-lockbox key*. The signature of the root is computed and verified using the *file-sign key* and *file-verify key*, respectively.

the file's owner. The filegroup and owner information is located in the inode, as in the case of UNIX. The names of files and filegroups can be encrypted with the *file-name key* and *filegroup-name key*, respectively. Encrypting the names of files and filegroups protects against attacks where the malicious user can glean information about the nature of the file.

All the above described keys are generated and distributed by the owners of the files and filegroups. In addition, currently in Plutus, readers and writers can (re)distribute the keys they have to other users. Plutus intentionally avoids specifying the protocols needed to authenticate users or distribute keys: these are independent of the mechanisms used to secure the stored data and can be chosen by individual users to match their needs.

3.3 Read-write differentiation

One of the basic security functions that file systems support is the ability to have separate readers and writers to the same file. In Plutus, this cannot be enforced by the server as it itself is untrusted; instead we do this by the choice of keys distributed to readers and writers. File-lockbox keys themselves cannot differentiate readers and writers, but can do so together with the file-sign and file-verify key pairs. The file-sign keys are handed to writers only, while readers get the file-verify keys. When updating a data block, a writer recomputes the Merkle hash tree over the (current) individual block hashes, signs the root hash, and places the signed hash in the header of the file. Readers verify the signature to check the integrity of the blocks read from the server. Though using public/private keys for differentiated read/write access was mentioned in the work on securing replicated data [49], the design stopped short of finding a cryptosystem to implement it.

Note that though the file-verify key is same as the public key in a standard public-key system, it is not publicly disseminated. Owners of files issue the file-verify key only to those they consider as authorized readers; similar is the case with the file-sign key.

In our implementation, we use RSA for the sign/verify operations. Then only the readers and writers know N (the RSA modulus). The file-verify key, e, is not a low-exponent prime number (it has to be greater than $N^{1/4}$ [6]). Writers get (d, N), while readers get (e, N).

3.4 Lazy revocation

In a large distributed system, we expect revocation of users to happen on a regular basis. For instance, according to seven months of AFS protection server logs we obtained from MIT, there were 29,203 individual revocations of users from 2,916 different access control lists (counting the number of times a single user was deleted from an ACL). In general, common revocation schemes, such as in UNIX and Windows NT, rely on the server checking for users' group membership before granting access. This requires all the servers to store or cache information regarding users, which places a high trust requirement on the servers and requires all the servers to maintain this authentication information in a secure and consistent manner.

Revocation is a seemingly expensive operation for encrypt-on-disk systems as it requires re-encryption (in

Plutus, re-computing block hashes and re-signing root hashes as well) of the affected files. Revocation also introduces an additional overhead as owners now need to distribute new keys to users. Though the security semantics of revocation need to be guaranteed, they should be implemented with minimal overhead to the regular users sharing those files.

To make revocation less expensive, one can delay re-encryption until a file is updated. This notion of lazy revocation was first proposed in Cepheus [13]. The idea is that there is no significant loss in security if revoked readers can still read unchanged files. This is equivalent to the access the user had during the time that they were authorized (when they could have copied the data onto floppy disks, for example). Expensive re-encryption occurs only when new data is created. The meta-data still needs to be immediately changed to prevent further writes by revoked writers. We discuss subtle attacks that are still possible in Section 4.

A revoked reader who has access to the server will still have read access to the files not changed since the user's revocation, but will never be able to read data updated since their revocation.

Lazy revocation, however, is complicated when multiple files are encrypted with the same key, as is the case when using filegroups. In this case, whenever a file gets updated, it gets encrypted with a new key. This causes filegroups to get fragmented (meaning a filegroup could have more than one key), which is undesirable. The next section describes how we mitigate this problem; briefly, we show how readers and writers can generate all the previous keys of a fragmented filegroup from the current key.

3.5 Key rotation

The natural way of doing lazy revocation is to generate a new filegroup for all the files that are modified following a revocation and then move files to this new filegroup as files get re-encrypted. This raises two issues: a) there is an increase in the number of keys in the system following each revocation; and b) because the sets of files that are re-encrypted following successive revocations are not really contained within each other, it becomes increasingly hard to determine which filegroup a file should be moved to when it is re-encrypted. We address the first issue by relating the keys of the involved filegroups. To address the second issue, we set up the keys so that files are always (re)encrypted with the keys of the latest filegroup; then since keys are related users need to just remember the latest keys and derive previous ones when necessary. We call the latter process *key rotation*.

There are two aspects of rotating the keys of a filegroup a) rotating file-lockbox keys, and b) rotating file-sign and file-verify keys. In either case, to make the revocation secure, the sequence of keys must have the following properties:

a) Only the owner should be able to generate the next version of the key from the current version. This is to prevent anyone from undoing the revocation.
b) An authorized reader should be able to generate all previous versions of the key from the current version. Then readers maintain access to the files not yet re-encrypted, and readers may discard previous versions of the key.

In Plutus, each reader has only the latest set of keys. Writers are directly given the newest version of the keys, since all file encryptions always occur with the newest set of keys. The owners could also do the new-key distribution non-interactively [14], without making point-to-point connections to users.

To assist users in deciding which keys to use, each key has a version number and an owner associated with it. Each file has the owner information, and the version number of the encryption key embedded in the inode. Note that this serves only as a hint to readers and is not required for correctness. Readers can still detect stale keys when the block fails to pass the integrity test.

Next we will describe how we achieve key rotation for file-lockbox keys and file-sign/file-verify keys.

3.5.1 Rotating file-lockbox keys

Whenever a user's access is revoked, the file owner generates a new version of the file-lockbox key. For this discussion, let v denote the version of the file-lockbox key. The owner generates the next version file-lockbox key from the current key by exponentiating the current key with the owner's private key (d, N): $K_{v+1} = K_v^d \bmod N$. This way only the owner can generate valid new file-lockbox keys.

Authorized readers get the appropriate version of the file-lockbox key as follows. (Figure 2 illustrates the relation between the different file-lockbox key versions.) Let w be the current version of the file-lockbox key that a user has.

- If $w = v$ then the reader has the right file-lockbox key to access the file.
- If $w < v$ then the reader has an older version of the key and needs to request the latest file-lockbox key from the owner.
- If $w > v$ then the reader needs to generate the older version of the file-lockbox key using the following recursion. If K_w is the file-lockbox key associated with version w, then $K_{w-1} = K_w^e \bmod N$, where (e, N) is the owner's public key. Readers can recursively generate all previous file-lockbox key from the current key.

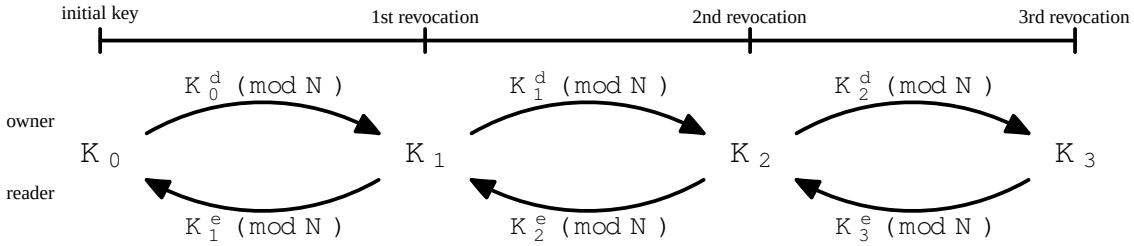

Figure 2: Key rotation for file-lockbox keys. Using RSA, an owner can rotate a key K_i forward. Users can only rotate keys backwards in time. (e, N) is the owner's public key and (d, N) is the owner's private key

In the above protocol, we use RSA encryption as a pseudorandom number generator; repeated encryption is not likely to result in cycling, for otherwise, it can be used to factor the RSA modulus N [33]. Though we use RSA for our key rotation, the property we need is that there be separate encryption and decryption keys, and that the sequence of encryptions is a pseudorandom sequence with a large cycle; most asymmetric cryptosystems have this property.

Though this scheme resembles Lamport's password scheme [27], our scheme is more general. Our scheme provides for specific users (owners) to rotate the key forward, while allowing some other users (readers) to rotate keys backwards.

3.5.2 Rotating file-sign and file-verify keys

By using the file-lock box key generated above as a seed, we can bootstrap the seed into file-sign and file-verify keys as follows. Let the version v file-sign key be (e_v, N_v) and the corresponding file-verify key be (d_v, N_v). In Plutus N_v is stored in file's header in the clear, signed by the owner to protect its integrity. Note that all files in the file-group with the same version have the same value for N_v.

When a user is revoked, the owner picks a new RSA modulus N_v, and rotates the file-lockbox key forward to K_v. Using the latest seed K_v, owners and readers generate the file-verify key as follows. Given the seed K_v, e_v is calculated by using K_v as a seed in a pseudo-random number generator. The numbers output are added to $\sqrt{N_v}$ and tested for primality. The first such number is chosen as e_v. The conditions that $e_v \geq \sqrt{N_v}$ and e_v is a prime guarantee that $\gcd(e_v, \phi(N_v)) = 1$ [28], making it a valid RSA public key. (Notice that the latter test cannot be performed by readers because they do not know $\phi(N_v)$). The pair (e_v, N_v) is the file-verify key.

Owners generate the corresponding RSA private key d_v and use it as the file-sign key. Since writers never have to sign any data with old file-sign keys, they directly get the latest file-sign key (d_v, N_v) from the owner. If the writers have no read access, then they never get the seed, and so it is hard for them to determine the file-verify key from the file-sign key.

Given the current version seed K_v, readers can generate previous version file-verify keys (d_u, N_u), for $u < v$ as follows. They first rotate the seed K_v backwards to get the seed K_u, as described in the previous section. They read (and verify) the modulus N_u from the file header. They then use the procedure described above to determine e_u from N_u and K_u.

The reason for changing the modulus after every revocation is to thwart a subtle collusion attack involving a reader and revoked writer – if the modulus is fixed to, say N', a revoked writer can collude with a reader to become a valid writer (knowing e_v, d_v, and N' allows them to factor N', and hence compute the new file-sign key).

3.6 Server-verified writes

The final question we address is how to prevent unauthorized writers from making authentic changes to the persistent store. Because the only parties involved in the actual write are the server and the user who wishes to write, we need a mechanism for the server to validate writes.

In traditional storage systems, this has been accomplished using some form of an access control list (ACL); the server permits writes only by those on the ACL. This requires that the ACL be stored securely, and the server authenticates writers using the ACL.

In Plutus, a file owner stores a hash of a write token stored on the server to validate a writer. This is semantically the same as a shared password.

Suppose a filename F is not encrypted. The owner of the file creates a write-verification key K_w as the write token. Then, F and the hash of the write token, $H[K_w]$, are stored on the server in the file's inode.

Upon authentication, writers get the write token K_w from the owner. When writers issue a write, they pass the token to the server. To validate the write, the server can now compute $H[K_w]$ and compare it with the stored value. Readers cannot generate the appropriate token because they do not know K_w. The token is secure since the hash value is stored only on the server. Optionally the

server can cache the hashed write tokens to speed up write verification.

One problem with the above scheme is that from $H[K_w]$, anyone can learn useful structural information such as which files belong to which filegroup even when the filegroup name is encrypted. This is undesirable given that storage system itself can be stolen and it does not do any authentication of the readers. Such attacks can be thwarted by replacing $H[K_w]$ with $H[K_w, F]$ and the filegroup name with $H[K_g, F]$, where K_g is the filegroup-name key.

The write token used above is similar to the capabilities used in NASD [19] and many systems before [29]. However capabilities in general are given out by a centralized server whereas write tokens are generated by individual file owners and are given to writers in a distributed manner.

The benefit of this approach is that it allows an untrusted server to verify that a user has the required authorization, without revealing the identity of the writer to the server. The scheme also makes it easy for the server to manage the storage space by decoupling the information required to determine allocated space from the data itself. Though the actual data and (possibly) filenames and filegroup names are encrypted and hidden, the list of physical blocks allocated is visible to the server for allocation decisions.

There are several file systems such as Cedar [18], Elephant [41], Farsite [1], Venti [38], and Ivy [36], which treat file data as immutable objects. In a cryptographic storage file system with versioning, server-verified writes are less important for security. Readers can simply choose to ignore unauthorized writes, and servers need worry only about malicious users consuming disk space. In non-versioning systems, a malicious user could corrupt a good file, effectively deleting it.

4 Security analysis

This section explores the set of attacks that remain possible and explains how to adapt Plutus to thwart these attacks. We also argue that some of the remaining attacks can never be handled within the context of our system at any reasonable additional cost.

In decreasing order of severity, an attacker may:

(a) write new data with a new key
(b) write new data with an old key
(c) write old data with an old key; that is, revert to an old version
(d) destroy data
(e) read updated data
(f) read data that has not yet been updated.

These attacks can be prevented by some combination of the following mechanisms: change the read/write verification token (T), re-encrypt the lockbox with a new key (L), and re-encrypt the file itself with a new key (D). Table 1 presents the possible attacks classified into those that a revoked reader could mount, or those that a revoked writer could mount. In each case, the attacker may act alone or in collusion with the server. The attacks that writers can mount depend upon whether an unsuspecting reader has the updated keys or not.

If a system uses lazy revocation, we can prevent revoked readers from accessing data that has been updated. However to prevent them from accessing data that has not been updated, we would need some form of "read verification" — verification of read privileges on each read access, analogous to write-verification. If this verification were done by the storage server then the reader could not get to the data alone, but could do so in collusion with the server. To prevent this attack, the file must be re-encrypted, re-encrypting just the lockbox would be insufficient.

The problem with revoked writers is more severe. Again, we can prevent revoked writers from updating data by verifying each write. But if this verification is done by the server – as in server-verified writes – the system is subject to an attack by a revoked writer colluding with the server to make valid modifications to data. The only way to prevent this would be to broadcast the changed key to all users aggressively . Otherwise, a revoked writer will always be able to create data that looks valid and cheat (unsuspecting) readers who have not updated their key.

From the above discussion, it should be clear that lazy revocation is always susceptible to attacks mounted by revoked users in collusion with the server, unless a third (trusted) party is involved in each read and write access.

Finally, the server could mount the following attack, which we consider very difficult for the system to handle. In a *forking attack* [31], a server forks the state of a file between users. That is, the server separately maintains file updates for the users. The forked users never see each other's changes, and each user believes its state reflects reality. A higher level Byzantine agreement protocol, which is potentially expensive, might be necessary to address this issue [11]. Recently Mazières and Shasha [32] introduced the notion of *fork consistency* and a protocol to achieve it. Though their scheme does not prevent a forking attack, it makes it easier to detect.

5 Protocols

We now summarize the steps involved in protocols for creating, reading and writing as well as revoking users. We would like to remark again that all the keys and to-

Users	Key freshness	Collusion	None	D	L	LD	T	TD	TL	TLD
revoked reader	old keys	alone	f	f	f	–	–	–	–	–
		w/ server	c,d,f	c,d,f	c,d,f	c,d,f	c,d,f	c,d	c,d,f	c,d,f
revoked writer	old keys	alone	c,b,d	c,b,d	c,b,d	c,b,d	–	–	–	–
		w/ server	c,b,d	c,b,d	c,b,d	c,b,d	c,b,d	c,b,d	c,b,d	c,b,d
	updated keys	alone	n/a	n/a	d	d	n/a	n/a	–	–
		w/ server	n/a	n/a	d	d	n/a	n/a	d	d

Table 1: Attacks tabulated against what is changed following a revocation. The heading row presents different choices in the component that is changed following a revocation: the read/write verification token is changed (T), the file's lockbox is changed (L), or the file itself is re-encrypted with a new key (D). The entries in the table correspond to the most serious attack that can be mounted, the letter code corresponding to those described in the main text. "n/a" indicates an impossible combination – such as readers having updated keys but files not being re-encrypted or lockboxes not changed. A "–" is used to denote that no attack is possible.

kens in these protocols are exchanged between owners and readers/writers via a secure channel with a session key – for instance, mutual authentication based on passwords. However, file data is not encrypted over the wire, but only integrity-protected with the session key.

1. **Initialize filegroup:** To initialize a filegroup, a user generates a pair of dual keys (file-sign and file-verify keys) for signing and verifying the contents of files in the filegroup. The user also generates the symmetric file-lockbox key.

2. **Create file:** First, the owner selects a filegroup for the new file. If there is no appropriate filegroup the owner initializes one and uses the corresponding keys (file-sign, file-verify, and file-lockbox keys) for this file. The owner also generates a write token and sends it to the server so that the server can verify all writes to this file.

3. **Read file:** A reader first obtains the name of the owner and the filegroup of the file he wishes to access, possibly after browsing the file system. The reader then checks if the version of the keys she has cached is greater then the version of the keys used to encrypt the file (which is stored in the header), in which case she does a key rotation to get the right version key. Otherwise, the reader gets the latest version key from the owner after appropriate authentication (via a secure channel). The reader then fetches the encrypted blocks of the desired file from the server, opens the lockboxes with the file-lockbox key, retrieves file-block keys from the lockbox, and decrypts the individual blocks. The integrity of the root hash (of the Merkle tree) provided by the server is first verified by using the file-verify key. To verify the integrity of the data, this root hash is compared against the root hash obtained by recomputing the Merkle hash tree using the file blocks retrieved from the server.

4. **Write file:** The writer obtains the latest version file-lockbox key and file-sign key, possibly from the owner if it is not cached. The writer then generates the file-block keys, encrypts individual blocks of the file using the corresponding file-block keys, and stores the encrypted blocks with the lockbox in the server. The server uses the write-token, provided by the writer at this time, to authorize the write. The writer then sends the entire Merkle hash tree in the clear to the server; the hash tree includes the root hash, signed with the file-sign key.

5. **Revoke user:** To revoke a user from accessing files in a filegroup, the owner generates the next version of the file-sign, file-verify, and file-lockbox keys. The owner then labels all files in the filegroup as needing re-encryption. If the revoked user is a writer, the owner changes the write-tokens in all the files of the filegroup as well.

6 Implementation

Using the protocols and ideas discussed in Section 3, we have designed Plutus and developed a prototype using OpenAFS [37]. In this section, we describe the architecture and the prototype of Plutus in detail.

6.1 Architecture of Plutus

Figure 3 summarizes the different components of Plutus, and where (server side or client side) they are implemented. Both the server and the clients have a network component, which is used to protect the integrity of all messages. In our implementation we protect the integrity of packets in the AFS RPC using HMAC [4]. Additionally, some messages such as those initiating read and write requests are encrypted. A 3DES session key, exchanged as part of the RPC setup, is optionally used to encrypt

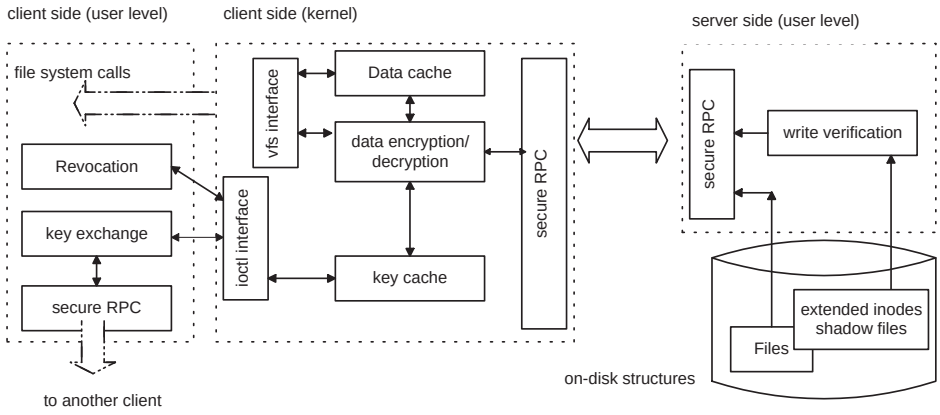

Figure 3: Architecture of Plutus.

these packets. The identities of all entities are established using 1024 bit RSA public/private keys.

The server has an additional component that validates writes. As described in Section 3.6, this component computes the SHA-1 hash of write tokens to authorize writes. This hashed token is passed on to the server, when the file is created, and is stored in the file's vnode (UNIX inode extension in AFS). Storing the token in the vnode instead of the directory simplifies write verification[2]. Owners change the stored token using a special ioctl call.

Most of the complexity of the implementation is at the client-side. We extended the OpenAFS kernel module by adding a key cache per user and a component to handle file data encryption and decryption. The key cache holds all keys used by the user, including file keys and identity keys (users' and servers' public keys). Currently all the encryptions and decryptions are done below the AFS cache; that is, we cache clear-text data. By doing this we encrypt (decrypt) file contents only when it is being transmitted to (received from) the server. The alternative of caching encrypted data would mean that each partial read/write would incur a block encryption/decryption, as would multiple reads/writes of the same block. We expect this to incur a substantial cryptographic overhead. Of course, caching unencrypted data opens up a security vulnerability on shared machines.

The other components of the client – revocation and key exchange – are implemented in user space. These components interact with the key cache through an extension to AFS's ioctl interface. The same client-server RPC interface is used for all inter-client communication.

Files are fragmented, and each fragment (blocks of size 4 KB) is encrypted independently with its own file-block (3DES) key. This 3DES key is kept in the fragment's lockbox together with the length of the fragment. The hashes of all the fragments are arranged in a Merkle hash tree, and the root signed (1024 bit RSA) with the file-sign-key. The leaves of the tree contain the lockbox of the corresponding fragment. The tree is kept in a "shadow file," on the server, and is shipped to the client, when the corresponding file is opened. On the client side, when blocks are updated, the respective new hashes are spliced into the tree. Then, the root hash is recomputed and signed when the cache is to be flushed to the server. At this time, the new tree is also sent back to the server.

6.2 Prototype

In building the Plutus prototype, we have made some modifications to the protocols to accommodate nuances of AFS. However, these modifications have little impact on the actual evaluation reported in the next section. For instance, currently AFS's RPC supports only authentication of the client by the server through a three step procedure. Recall that in Plutus design, the server never needs to authenticate a client. We use only the last two steps of this interface to achieve reverse authentication (i.e., client authenticating server) and session key exchange. To do this we need the server's public key, which can be succinctly implemented with self certifying pathnames [30], thus securely binding directories to servers.

The prototype uses a library that was built from the cryptographic routines in GnuPGP, with the following choice of primitives: 1024-bit RSA PKCS#1(version 1.5)[3] for public/private key encryption, SHA-1 for hashing and 3DES with CBC with Cipher Text Stealing [45] for file encryption.

[2] A similar problem was encountered in the context of storing inodes and small files together [16].

[3] For better security guarantees, RSA-OAEP is required; see Shoup's proposal [46] for more details.

Appears in the Proceedings of the 2nd Conference on File and Storage Technologies (FAST'03). pp. 29–42 (31 Mar – 2 Apr 2003, San Francisco, CA). Published by USENIX, Berkeley, CA

7 Performance evaluation

In the preceding sections, we analyzed protocols of Plutus from a security perspective. We now evaluate Plutus from a performance perspective. In particular, we evaluate the design and the prototype of Plutus using (a) a trace from a running UNIX system, and (b) synthetic benchmarks. Using (a), the trace statistics, we argue the benefits of filegroups and the impact of lazy revocation. By measuring the overhead of Plutus using (b), synthetic benchmarks, we argue that though there is an overhead for the encryption/decryption, Plutus is quite practical; in fact, it compares favorably with SFS.

7.1 Trace evaluation

The trace that we use for evaluation is a 10-day complete file system trace (97.4 million requests, 129 GB data moved and 24 file systems) of a medium-sized workgroup using a 4-way HP-UX time-sharing server attached to several disk arrays and a total of 500 GB of storage space. This represents access to both NFS filesystems that the server exported, and accesses to local storage at the server. The trace was collected by instrumenting the kernel to log all file system calls at the syscall interface. Since this is above the file buffer cache, the numbers shown will be pessimistic to any system that attempts to optimize key usage on repeated access.

Key sharing	System		User	
	mean	max	mean	max
key/file	1,700	9,200	900	41,100
key/filegroup	11	57	6	23

Table 2: Using filegroups to aggregate keys.

KEYS AND FILEGROUPS

Table 2 presents the number of keys distributed among users. We classified all the user-ids in the system into System (such as root, bin, etc.) and User (regular users). The first row represents the number of keys that need to be distributed if a different key is used for each file in the system; the second row represents the number of keys distributed if filegroups are used. In this evaluation, we used the (mode bits, owner, group) tuple to define filegroups. The table presents numbers for both the maximum number of keys distributed by any user, and the mean number of keys distributed (averaged across all users who distributed at least one key). The table demonstrates the benefit of using filegroups clearly: the maximum number of keys distributed is reduced to 23, which is easy to manage. Note that even this is a pessimistic evaluation as it assumed a *cold key cache*.

OVERHEAD OF REVOCATION

Table 3 presents parameters of the traced system that affect the overhead of performing a revocation. In this context we focus on the case where the owner of a filegroup wants to revoke another user's permission to read/write files in the owner's filegroup. We use these parameters to evaluate the overhead of performing a revocation, both in terms of carrying out the operations immediately following a revocation, and re-distributing the updated keys to other users. In the case of revoking a reader, the time spent immediately following a revocation is the time required to mark all files in the filegroup as "to be re-encrypted." In the case of revoking a writer this is the time to change the write verification key of all the files in the filegroup. For the system we traced, if a user revokes another user, this would involve marking 4,800 files to be re-encrypted, on average, and about 119,000 files, maximum. When a user (reader or writer) is revoked, other users (readers/writers) need to be given the updated key. Our evaluation shows that this number is typically very small: 2 on average and at most 11 in the worst case.

7.2 Cryptographic cost

Table 4 presents the impact of encryption/decryption on read and write latency. These are measurements of the cryptographic cost that includes write verification, data encryption, and wire-transmission overheads. These were done using code from Plutus' cryptography library on a 1.26 GHz Pentium 4 with 512 MB memory. In this evaluation, we used 4 KB as the size of the file fragment (corresponding to that of the prototype). As in the prototype, for data encryption, we used 1,024-bit RSA with a 256-bit file-verify key for reading and a 1,019-bit file-sign key for writing and 3DES CBC/CTS file-block key for bulk data encryption.

Owners incur a high one time cost to generate the read/write key pair; this is another reason why aggregating keys for multiple files using filegroups is beneficial. Though the write verification latency is negligible for writers and owners, if we choose to hide the identities of filegroups, then we pay an additional cost of decrypting it. The time spent in transmitting the Merkle hash tree depends on the size of the file being transmitted. In Plutus, block hashes are computed over 4 KB blocks, which contribute to about 1% overhead in data transmission.

For large files, the block encryption/decryption time dominates the cost of writing/reading the entire file. Though Plutus currently uses 3DES as the block cipher, from Dai's comparison of AES and 3DES [9], we expect a 3X speedup if AES were used.

Parameters	User			System		
	Highest	Second	Mean	Highest	Second	Mean
number of files	119,000	101,200	4,800	1,561,000	94,000	29,800
total bytes	17 GB	11 GB	0.6 GB	29 GB	14 GB	1.3 GB
number of readers	5	4	1.2	27	22	5.4
number of writers	6	5	0.7	15	14	1.7

Table 3: Parameters of the system that affect revocation. These are statistics indicating the number of files in a single filegroup owned by a user, the total size of all these files, the number of other users who have read permission to at least one of these files, and the number of other users who have write permission to at least one of these files. The number of readers and writers were determined by considering the accesses in the 10-day trace, while the static information was gathered by considering a snapshot of the filesystem taken at the end of the 10 days. The table separates statistics for regular users and system users.

File system operation	Crypto operation	Crypto cost	Incurred by	Frequency
Filegroup creation	RSA key generation	2500 ms	owner	per filegroup
File write	Block hash	0.11 ms	writer	per 4 KB block
	Block encrypt	0.59 ms	writer	per 4 KB block
	Merkle root sign	28.5 ms	writer	per file
	Write verify	0.01 ms	server	per file
File read	Block hash	0.11 ms	reader	per 4KB block
	Block decrypt	0.61 ms	reader	per 4 KB block
	Merkle root verify	8.5 ms	reader	per file
Wire integrity	Message encrypt	0.01 ms	all	per 100 byte message
	Message decrypt	0.01 ms	all	per 100 byte message
	Message hash	0.003 ms	all	per 100 byte message

Table 4: Cryptographic primitive cost. This table lists the cost of the basic cryptographic primitives, and the file systems operations where they are incurred. The root signature and verification is done only once per file read or write, irrespective of the size of the file. Wire integrity is needed only for messages, not for file contents.

7.3 Benchmark evaluation

We used a microbenchmark to compare the performance of Plutus to native OpenAFS and to SFS [30]. The microbenchmark we used is modeled on the Sprite LFS large file benchmarks. These involve reading and writing multiple 40 MB files with sequential and random accesses.

We used two identically configured machines, as in the previous section, connected with a Gigabit ethernet link. In all these experiments we restarted the client daemon, before reading/writing any file. We present the mean of 6 out of 10 runs, ignoring the top and bottom two outliers.

Table 5 presents the results of this evaluation. First, the table shows that the overhead of Plutus is primarily dependent on the choice of block cipher used. For instance, it takes 5.9s to decrypt 40MB with 3DES, which is about 75% of the average sequential read latency. Thus Plutus with no-crypto is faster than that with DES, which is in turn faster than with 3DES.

Second, Plutus performs as well as (if not better than) the other two encrypt-on-wire systems. In these comparisons it is important to compare systems that use block ciphers with similar security properties. In particular, the performance of Plutus with DES is slightly better than that of OpenAFS with fcrypt as the cipher: the fcrypt cipher is similar to DES. Though Plutus with 3DES is about 1.4 times slower than SFS, the latter uses ARC4, which is known to have a throughput about 14 times that of 3DES [9]. This leads us to believe that if Plutus were modified to use ARC4 or AES, it would compare well with SFS.

Note that this experiment is a pessimistic comparison between Plutus and the other two encrypt-on-wire systems. In the setting where there are several clients accessing data from the same server, Plutus would provide better server throughput because the server does not perform much crypto. This would translate to lower average latencies for Plutus.

8 Related work

Most file systems including those in MS Windows, traditional UNIX systems, and secure file systems [19, 22, 30] do not store files encrypted on the server. Of course, the user may decide to encrypt files before stor-

File systems	Crypto options	Read		Write	
		seq	rand	seq	rand
Plutus	w/ 3DES cipher	7.84 s	7.78 s	7.92 s	8.13 s
	w/ DES cipher	4.58 s	4.54 s	4.27 s	4.79 s
	w/o crypto	1.39 s	1.51 s	1.59 s	2.64 s
OpenAFS	w/o wire-crypto	1.28 s	1.31 s	1.57 s	1.67 s
	w/ wire-crypto	4.66 s	4.90 s	5.34 s	5.43 s
SFS	w/ crypto	5.55 s	5.30 s	4.47 s	7.21 s

Table 5: Performance of Plutus, OpenAFS (version 1.2.8) and SFS (version 0.7.2) accessing 40 MB files with random and sequential access. The crypto option for Plutus indicates the cipher used for block encryption; the OpenAFS crypto option indicates whether it uses wire-crypto or not; SFS uses wire-crypto. OpenAFS uses fcrypt [3] for block encryption whereas SFS uses ARC4 [23]. The version of Plutus w/o crypto still performed all the operations required to manage and maintain the Merkle hash tree; the results indicate that this overhead is small.

age but this overwhelms the user with the manual encryption/decryption and sharing the file with other users – while trying to minimize the amount of computation. This is precisely the problem that Plutus addresses.

Though MacOS X and Windows CIFS offer encrypted disks, they do not allow group sharing short of sharing a password.

8.1 Secure file systems

In encrypt-on-disk file systems, the clients encrypt all directories and their contents. The original work in this area is the Cryptographic File System (CFS) [5], which used a single key to encrypt an entire directory of files and depended on the underlying file system for authorization of writes. Later variants on this approach include TCFS [8], which uses a lockbox to protect only the keys, and Cryptfs [51]. Cepheus [13] uses group-managed lockboxes with a centralized key server and authorization at the trusted server. SNAD [35] also uses lockboxes and introduces several alternatives for verifying writes. The SiRiUS file system layers a cryptographic storage file system over heterogenous insecure storage such as NFS and Yahoo! Briefcase [21].

Encrypt-on-wire file systems protect the data from adversaries on the communication link. Hence all communication is protected, but the data is stored in plaintext. Systems that use encryption on the wire include NASD (Networked Attached Storage) [20], NFS over IPSec [24], SFS (Self-Certifying File System) [30], iSCSI [42], and OpenAFS with secure RPC.

In these systems the server is trusted with the data and meta-data. Even if users encrypt files, the server knows the filenames. This is not acceptable if the servers are untrustworthy, as in a distributed environment where servers can belong to multiple administrative domains. On the positive side, this simplifies space management because it is easy for the server to figure out the data blocks that are in use. A comprehensive evaluation of these systems appear in a previous study [40].

8.2 Untrusted servers

One way to recover from a malicious server corrupting the persistent store is to replicate the data on several servers. In the state machine approach [26, 44], clients read and write data to each replica. A client can recover a corrupted file by contacting enough replicas. The drawback to this method is that each replica must maintain a complete copy of the data.

Rabin's Information Dispersal Algorithm divides a file into several pieces, one for each replica [39]. While the aggregate space consumed by all the replicas is minimal, the system does not prevent or detect corruption.

Alon et al. describe a storage system resistant to corruption of data by half of the servers [2]. A client can recover from integrity-damaged files as long as a threshold number of servers remain uncorrupted.

9 Conclusion

This paper has introduced novel uses of cryptographic primitives applied to the problem of secure storage in the presence of untrusted servers and a desire for owner-managed key distribution. Eliminating almost all requirements for server trust (we still require servers not to destroy data – although we can detect if they do) and keeping key distribution (and therefore access control) in the hands of individual data owners provides a basis for a secure storage system that can protect and share data at very large scales and across trust boundaries.

The mechanisms described in this paper are used as building blocks to design Plutus, a comprehensive, secure, and efficient file system. We built a prototype implementation of this design by incorporating it into OpenAFS, and measured its performance on micro-benchmarks. We

showed that the performance impact, due mostly to the cost of cryptography, is comparable to the cost of two popular schemes that encrypt on the wire. Yet, almost all of Plutus' cryptography is performed on clients, not servers, so Plutus has superior scalability along with stronger security.

Acknowledgements

We thank our shepherd Peter Honeyman for his comments and help while revising the paper. We thank Sameer Ajmani, Dan Boneh, Thomer Gil, Eu-Jin Goh, Dave Mazières, Chris Laas, Ethan Miller, Raj Rajagopalan, and Emil Sit for their helpful comments. We also thank Jeff Schiller, Garry Zacheiss, and Nickolai Zeldovich for producing logs of revocations on MIT's AFS protection servers.

References

[1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, pages 1–14, December 2002.

[2] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern. Scalable secure storage when half the system is faulty. In *ICALP*, 2000.

[3] T. Anderson. Specification of FCrypt: Encryption for AFS remote procedure calls,. http://www.transarc.ibm.com/∼ota/fcrypt-paper.txt.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, pages 1–15, 1996.

[5] M. Blaze. A cryptographic file system for UNIX. In *CCS*, 1993.

[6] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46, 1999.

[7] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *OSDI*, pages 273–288, October 2000.

[8] G. Cattaneo, G. Persiano, A. Del Sorbo, A. Cozzolino, E. Mauriello, and R. Pisapia. Design and implementation of a transparent cryptographic file system for UNIX. Technical report, University of Salerno, 1997.

[9] W. Dai. Crypto++ version 4.2. http://www.eskimo.com/∼weidai/cryptlib.html.

[10] C. Dalton and T. Choo. Trusted linux: An operating system approach. *CACM*, 44(2), February 2001.

[11] J. R. Douceur and R. P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *SRDS*, pages 4–13, 2001.

[12] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[13] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, MIT, June 1999.

[14] K. Fu, M. Kallahalla, S. Rajagopalan, and R. Swaminathan. Secure rotation on key sequences. Submitted for publication, 2002.

[15] K. Fu, M. Kallahalla, and R. Swaminathan. A simple protocol for maintaining fork consistency. Manuscript, 2002.

[16] G. Ganger and M. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Tech. Conf.*, pages 1–17, 1997.

[17] G. Ganger, P. Khosla, M. Bakkaloglu, M. Bigrigg, G. Goodson, S. Oguz, V. Pandurangan, C. Soules, J. Strunk, and J. Wylie. Survivable storage systems. In *DARPA Information Survivability Conference and Exposition, IEEE*, volume 2, pages 184–195, June 2001.

[18] D. Gifford, R. Needham, and M. Schroeder. The Cedar file system. In *CACM*, pages 288–298, March 1988.

[19] H. Gobioff, D. Nagle, and G. Gibson. Embedded security for network-attached storage. Technical Report CMU-CS-99-154, CMU, June 1999.

[20] H. Gobioff, D. Nagle, and G. A. Gibson. Integrity and performance in network attached storage. Technical Report CMU-CS-98-182, CMU, December 1998.

[21] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*, 2003. To appear.

[22] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM TOCS*, 6(1), February 1988.